.NET Full Stack Development Program

—

Day 13 ASP.NET Core MVC Fundamentals

Outline

- Fundamentals for ASP.NET Core MVC
  - Dependency Injection
  - Configuration
  - Middleware
- ASP.NET Core MVC 6.0 vs. before

Fundamentals for ASP.NET Core

- Dependency Injection
- Configuration
- Middleware

Dependency Injection - Loose Coupling

ASP.NET Core includes **dependency injection** (DI) that makes configured services available throughout an app. Services are added to the **DI container** with **WebApplicationBuilder.Services**

The built-in container is represented by **IServiceProvider** implementation that supports **constructor injection** by default

Dependency

You work in an organization where you and your colleagues tend to travel a lot. Your typical travel planning routine might look like the following:

- Decide the destination, and desired arrival date and time
- Call up the airline agency and convey the necessary information to obtain a flight booking.
- Call up the cab agency, request for a cab to be able to catch a particular flight
- Pickup the tickets, catch the cab and be on your way

Dependency

Now, what if your company suddenly changed the preferred agencies and their contact mechanisms?

You would be subject to the following relearning scenarios:

- The new agencies, and their new contact mechanisms (say the new agencies offer internet based services and the way to do the bookings is over the internet instead of over the phone)
- The typical conversational sequence through which the necessary bookings get done (Data instead of voice)

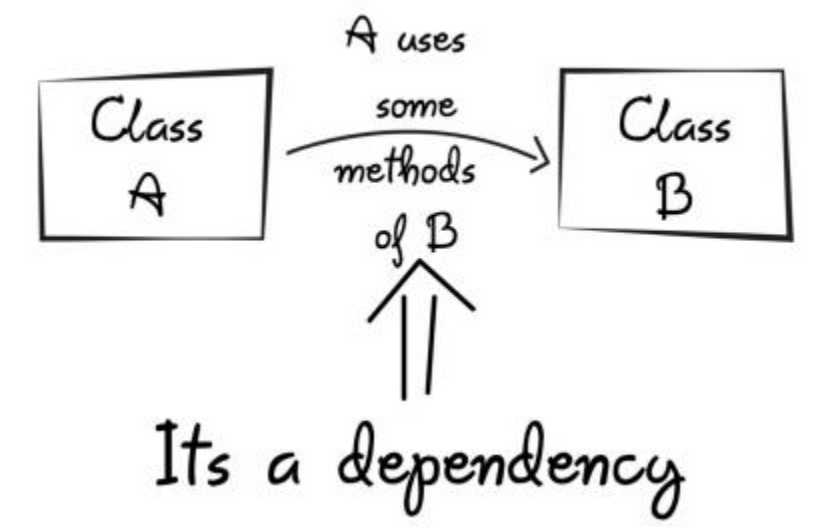
Dependency

Now let’s say the protocol is a little bit different.

- There is an administration department in the company
- Whenever you needed to travel an administration department interactive telephony system simply calls you up (which in turn is hooked up to the agencies)
- Over the phone you simply state the destination, desired arrival date and time by responding to a programmed set of questions
- The flight reservations are made for you, the cab gets scheduled for the appropriate time, and the tickets get delivered to you

Dependency

When class A uses some functionality of class B, then its said that class A has a dependency of class B.



Tightly Coupling

```
Public class Traveller {
    Car c = new Car();
    Public void startJourney() {
        c.move();
    }
}

Public class Car {
    Public void move(){
        ...
    }
}
```

```
Public class Traveller {
    Plane p = new Plane();
    Public void startJourney() {
        p.move();
    }
}

Public class Plane {
    Public void move(){
        ...
    }
}
```

Loose Coupling

```
Public class Traveller {
    Vehicle v;

    Public void setV(Vehicle v) {
        this.V = V;
    }
    Public void startJourney() {
        V.move();
    }
}
```

Dependency Injection

What is dependency Inject?

- Separating the usage from the creation of the object

Why dependency injection? (or Decoupling)

- It improves the testability
- It's much easier to swap other pieces of code/modules/objects/ components when the pieces are not dependent on one another
- Modularity — One module doesn't break other modules in unpredictable ways

IoC Container

- IoC — Inversion of Control
  - It is a process whereby objects define their dependencies (that is, the other objects they work with) only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method.
  - **Dependency injection** is a pattern through which to implement IoC, where the control being inverted is the setting of object's dependencies.
- IoC container injects those dependencies when it creates the instance

Service Lifetimes

Services can be registered with one of the following lifetimes:

- Transient
- Scoped
- Singleton

Service Lifetimes - Transient

Transient lifetime services are created each time they're requested from the service container.

```
builder.Services.AddTransient<Type>();
```

*Transient is considered the "**default**" service lifetime in .NET 6. This means that you should make all dependencies transient unless they truly need to use one of the other service lifetimes

Service Lifetimes - Scoped

A scoped lifetime indicates that services are created once per client request (connection)

```
builder.Services.AddScoped<Type>();
```

*When using **Entity Framework Core**, the **AddDbContext** extension method registers DbContext types with a **scoped lifetime by default.**

Service Lifetimes - Singleton

Singleton lifetime services are created either:

- The first time they're requested.
- By the developer, when providing an implementation instance directly to the container. This approach is rarely needed

*In apps that process requests, singleton services are disposed when the ServiceProvid

```
builder.Services.AddSingleton<Type>();
```

DI Container

Dependency Inject in .NET can be done in three ways:

- Constructor
- Setter (by default not supported, need built-in/3rd-party DI Container)
- Property (by default not supported, need built-in/3rd-party DI Container)

Constructor Injection

```
public class HomeController : Controller
{
    private readonly IDateTime _dateTime;

    public HomeController(IDateTime dateTime)
    {
        _dateTime = dateTime;
    }
```

```
services.AddSingleton<IDateTime, SystemDateTime>();
```

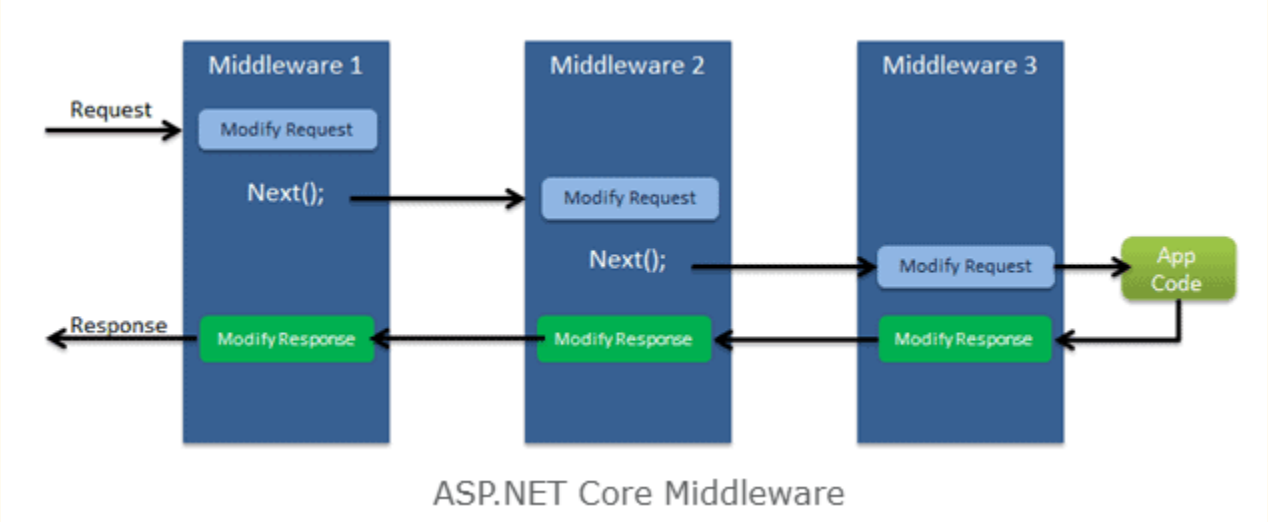
Configuration

We use **appsettings.json** file to configure application level settings, which can be used across the application and can be replaced the values at runtime without modifying the code.

- Load config data:
  - Load directly via **Configuration** object
    - Inject IConfiguration in the constructor in any class and then you can read all appsettings within this object
  - Bind hierarchical configuration data using the **options pattern**
    - The options pattern uses classes to provide strongly typed access to groups of related settings from appsettings.json file.

Middleware

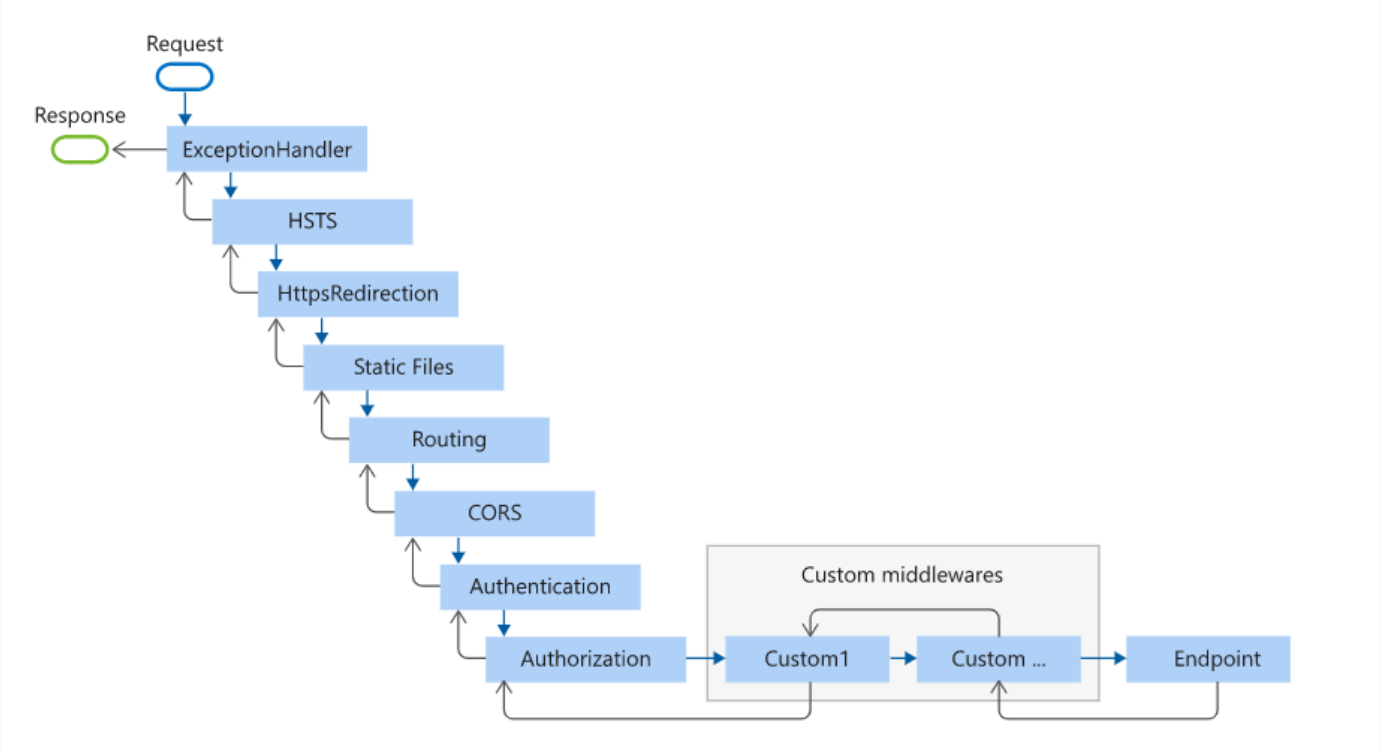
A **middleware** is nothing but a component (class) which is executed on every request in ASP.NET Core application



Middleware

- It has access to both Request and Response
- It may simply pass the Request to next Middleware
- It may process and then pass the Request to next Middleware
- It may handle the Request and short-circuit the pipeline
- It may process the outgoing Response
- Middlewares are executed in the order they are added to the pipeline

Middleware Order



Middleware

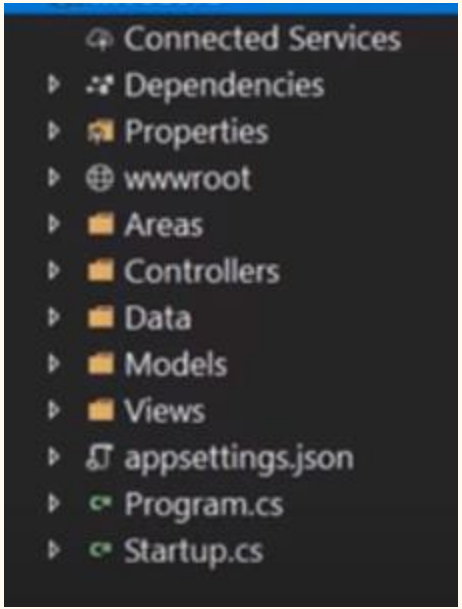
Everything after builder.build();

```
var builder = WebApplication.CreateBuilder(args);



// Add services to the container.
builder.Services.AddControllersWithViews();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    // The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();
```

Take a Look on ASP.NET Core MVC

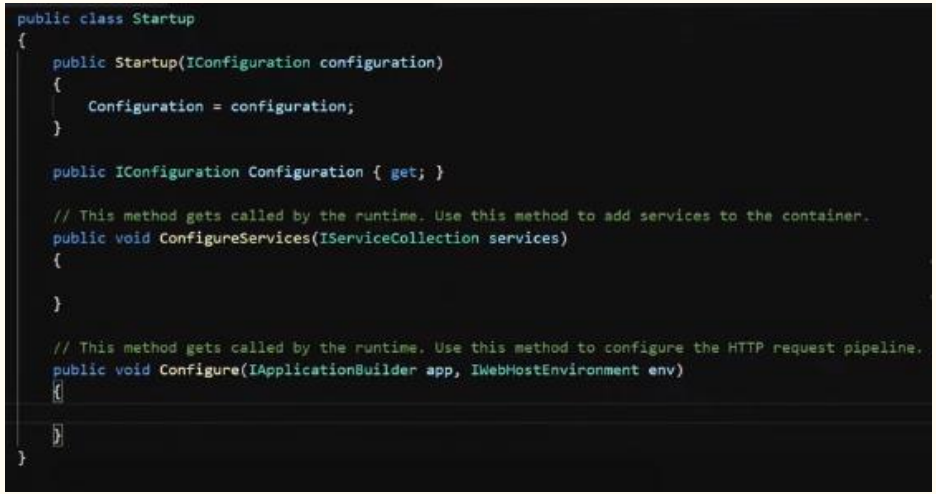
ASP.NET Core MVC – version before 6.0

- Wwwroot
  - Stores all the static files, ex. .css, .js, .html
- Areas
  - Identity system
- Startup.cs
  - Entry point. Setup configuration of how the app will be launched
- Program.cs
  - Where this application starts
- Appsettings.json
  - Define all the configs
  - Eg. config for dev, test, prod



Startup.cs

- ConfigureServices(IServiceCollection services)
  - register service for dependency injection
- Configure(IApplicationBuilder app, IWebHostEnviornment env)
  - outline your middleware pipeline order and structure



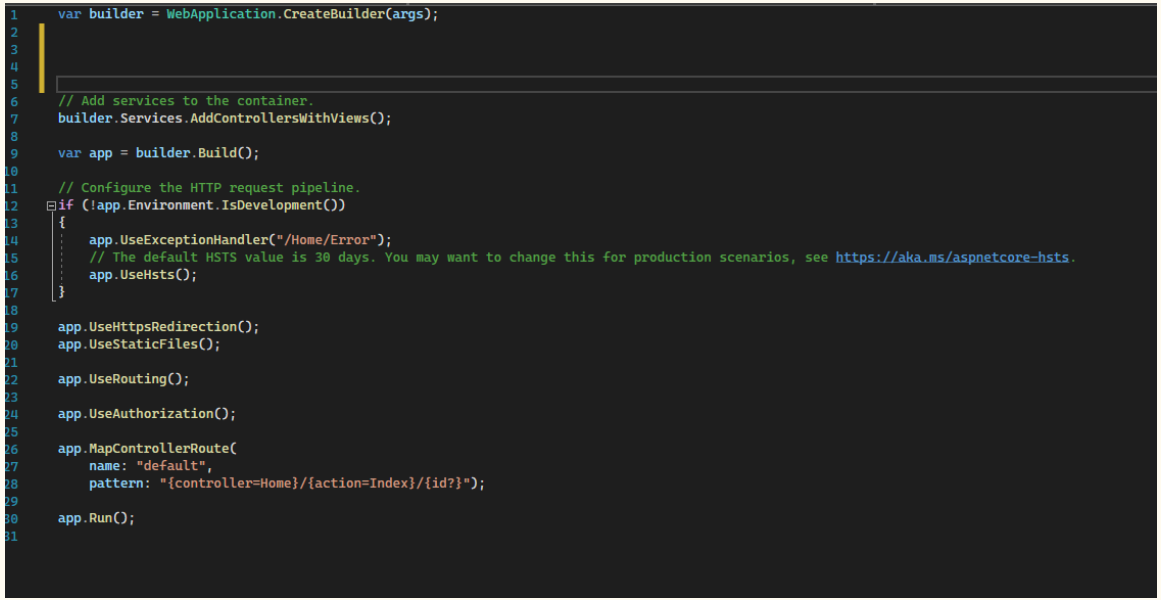
.NET 6.0

Putting all services for dependency injection between **builder** and **builder.Build().**
Eg. add Authentication, add dbConnection, and etc.

```
var builder = WebApplication.CreateBuilder(args);




// Add services to the container.
builder.Services.AddControllersWithViews();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    // The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();
```

.NET 6.0

In order to use those services, you need to put **app.UseXXX()** after **builder.Builder()**



```
var builder = WebApplication.CreateBuilder(args);




// Add services to the container.
builder.Services.AddControllersWithViews();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    // The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();
```

Question?